2020 자율주행 교육

WeGo 위고 주식회사

목차

01

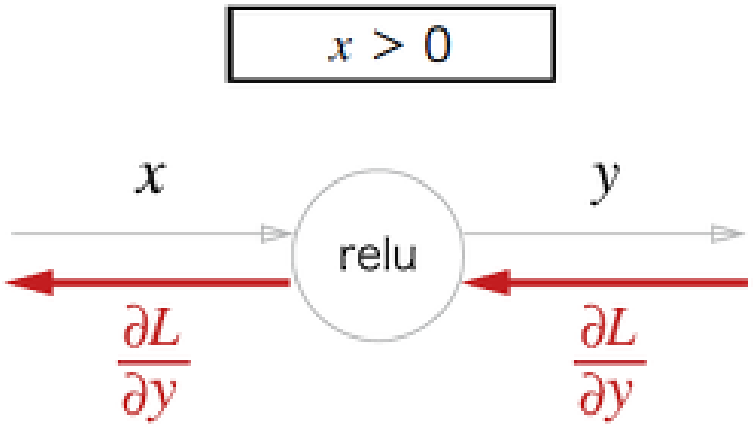
활성화 함수 계층 구현

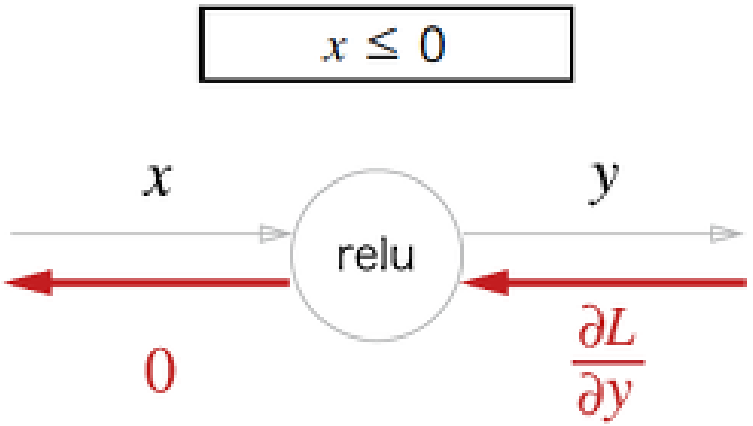
WeGo

- ReLU 계층 구현(Class 로 구현)

$$y = \begin{cases} x, & x < 0 \\ 0, & x \geq 0 \end{cases}$$

$$\frac{\partial y}{\partial x} = \begin{cases} 1, & x < 0 \\ 0, & x \geq 0 \end{cases}$$





01. 활성화 함수 계층 구현

- ReLU 계층 구현(Class 로 구현)

```
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

Mask의 경우, 입력에 따라,

True 및 False를 가지는 값을 출력

이 후, out[self.mask]를 통해

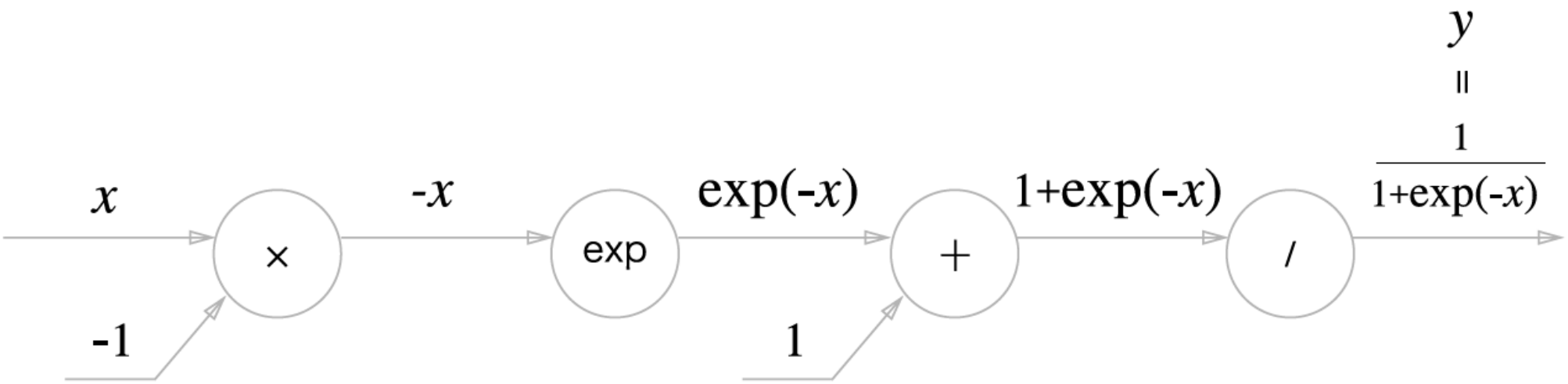
True인 부분만 0으로 처리

WeGo

01. 활성화 함수 계층 구현

- Sigmoid 계층 구현(Class 로 구현)

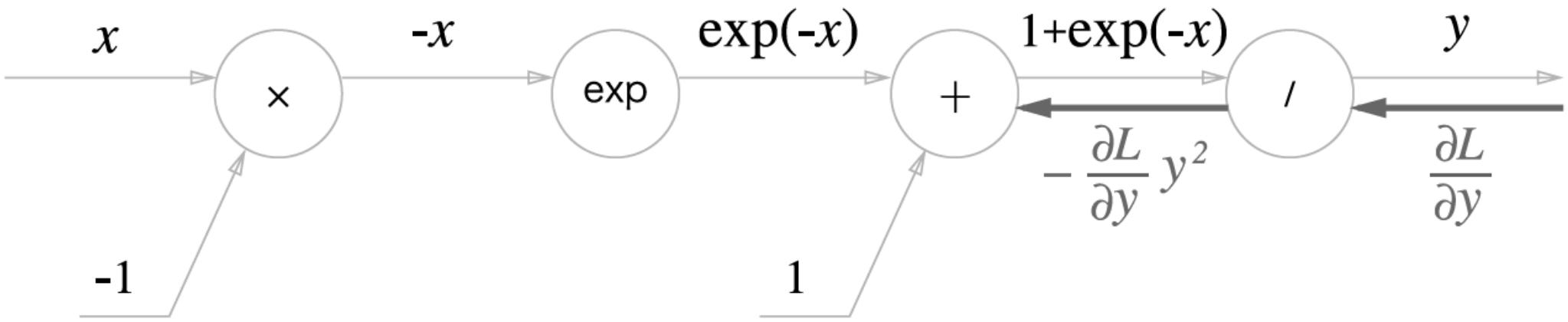
$$y = \frac{1}{1 + \exp(-x)}$$



01. 활성화 함수 계층 구현

- Sigmoid 계층 구현(Class 로 구현)

$$y = \frac{1}{x}, \frac{\partial y}{\partial x} = -\frac{1}{x^2} = -y^2$$

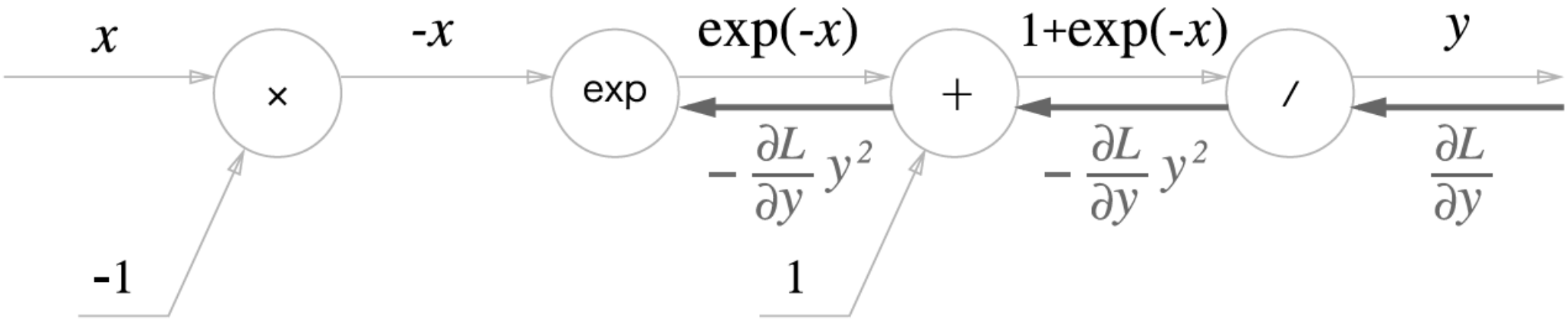


01. 활성화 함수 계층 구현

- Sigmoid 계층 구현(Class 로 구현)

$$y = \frac{1}{x}, \frac{\partial y}{\partial x} = -\frac{1}{x^2} = -y^2$$

덧셈 노드는 그대로 전달

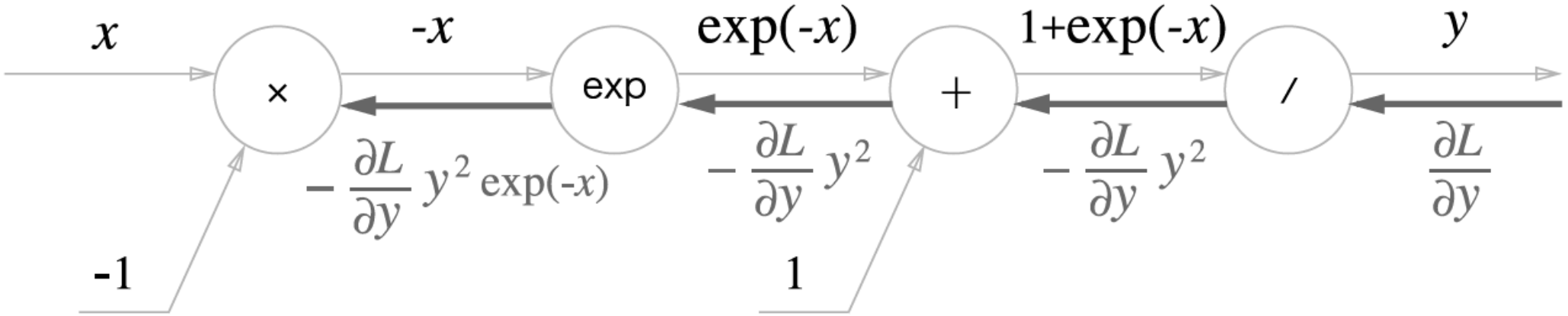


01. 활성화 함수 계층 구현

- Sigmoid 계층 구현(Class 로 구현)

$$\frac{\partial y}{\partial x} = \exp(x) \qquad y = \frac{1}{x}, \frac{\partial y}{\partial x} = -\frac{1}{x^2} = -y^2$$

덧셈 노드는 그대로 전달



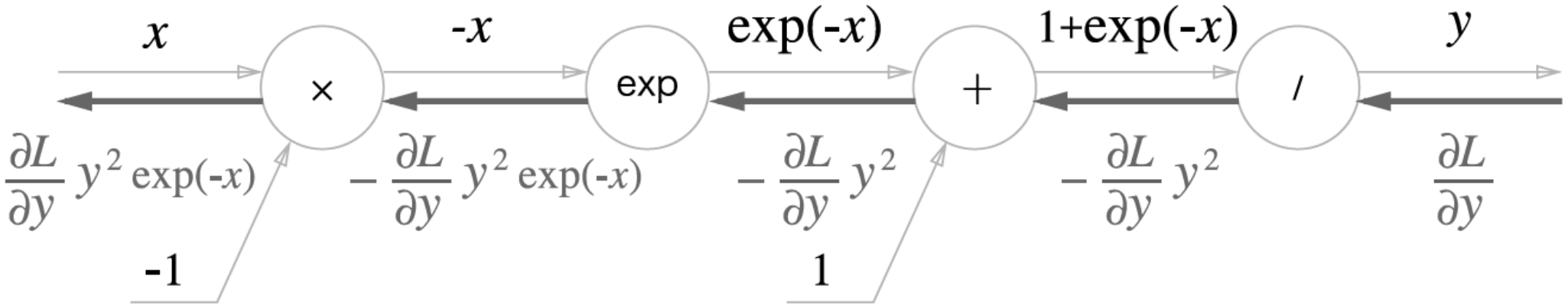
- Sigmoid 계층 구현(Class 로 구현)

$$\frac{\partial y}{\partial x} = \exp(x)$$

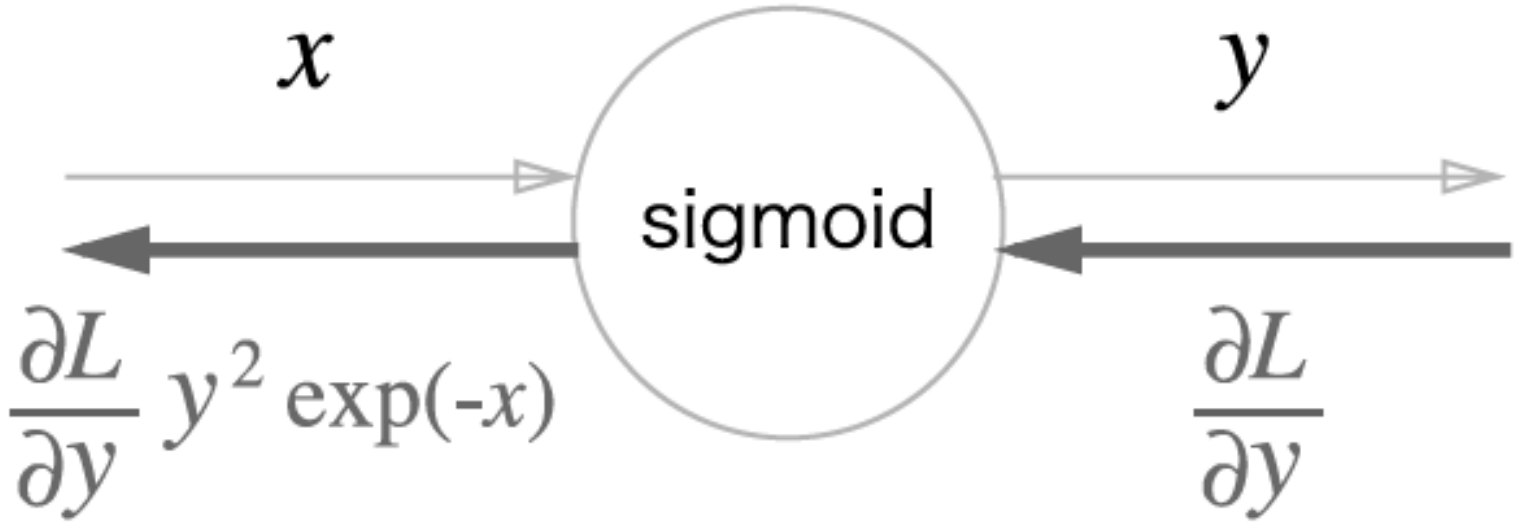
$$y = \frac{1}{x}, \frac{\partial y}{\partial x} = -\frac{1}{x^2} = -y^2$$

반대쪽에 해당하는 -1을 곱셈

덧셈 노드는 그대로 전달



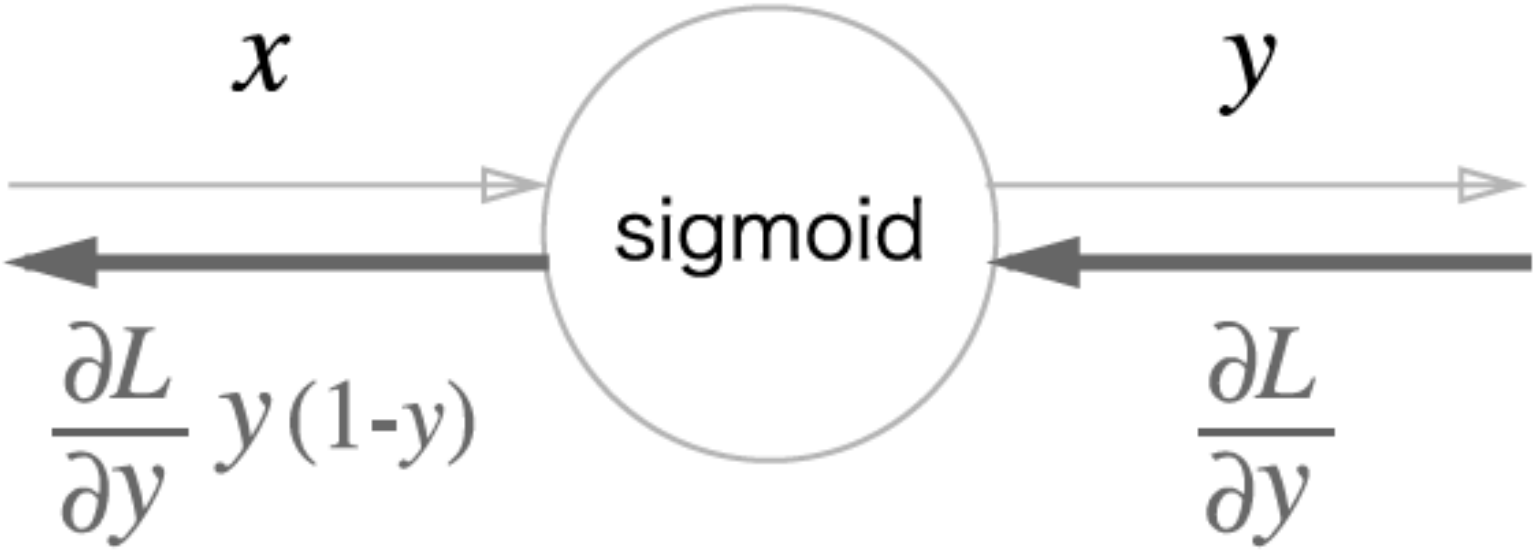
- Sigmoid 계층 구현(Class 로 구현)



- Sigmoid 계층 구현(Class 로 구현)

$$\frac{\partial L}{\partial y} y^2 \exp(-x) = \frac{\partial L}{\partial y} \frac{1}{(1+\exp(-x))^2} \exp(-x)$$

$$= \frac{\partial L}{\partial y} \frac{1}{(1+\exp(-x))} \frac{\exp(-x)}{(1+\exp(-x)} = \frac{\partial L}{\partial y} y(1-y)$$



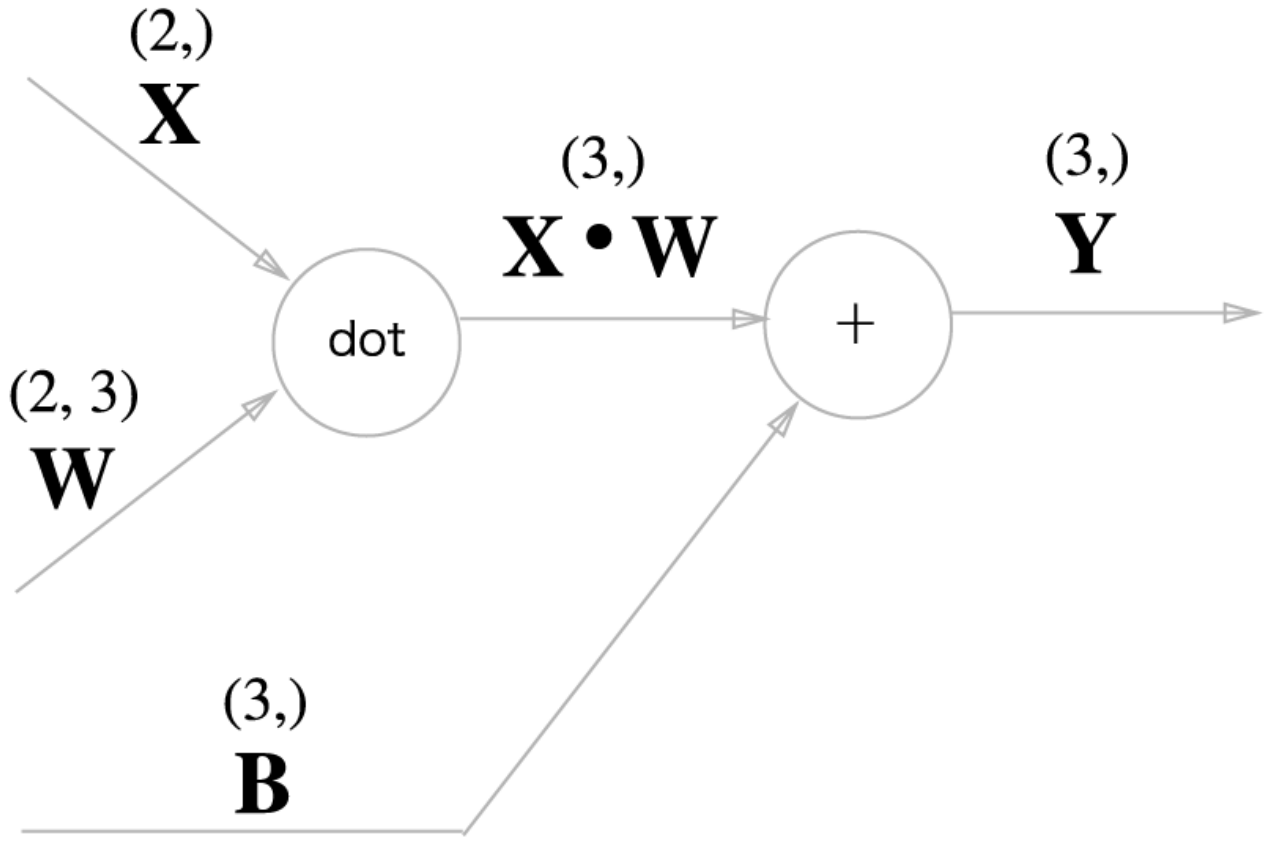
01. 활성화 함수 계층 구현

- Sigmoid 계층 구현(Class 로 구현)

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = sigmoid(x)
        self.out = out

        return out

    def backward(self, dout):
        dx = dout * (1.0 – self.out) * self.out

        return dx
```

01. 활성화 함수 계층 구현

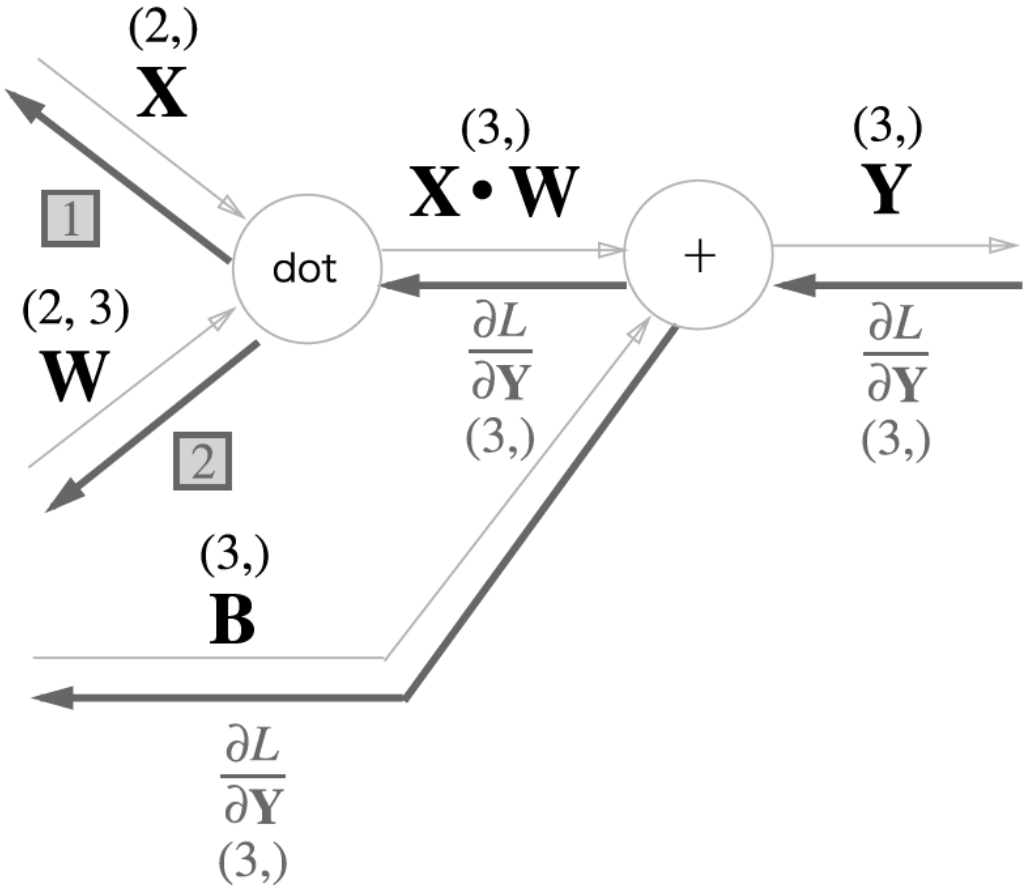
- Affine 계층 구현(Class 로 구현)
- 행렬의 곱을 기하학에서 Affine transformation이라 하여, Affine 계층이라고 함



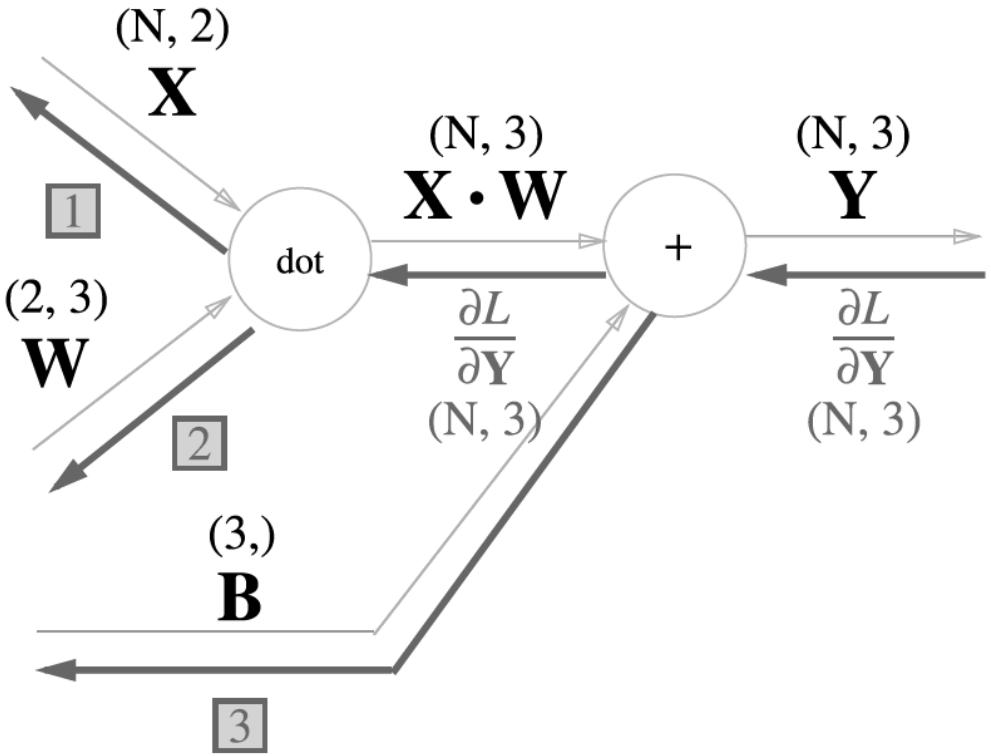
- Affine 계층 구현(Class 로 구현)

$$\boxed{1}\quad \underset{(2,)}{\frac{\partial L}{\partial \mathbf{X}}} = \underset{(3,)}{\frac{\partial L}{\partial \mathbf{Y}}} \underset{(3,\ 2)}{\mathbf{W}^{\mathrm{T}}}$$

$$\boxed{2}\quad \underset{(2,\ 3)}{\frac{\partial L}{\partial \mathbf{W}}} = \underset{(2,\ 1)}{\mathbf{X}^{\mathrm{T}}} \underset{(1,\ 3)}{\frac{\partial L}{\partial \mathbf{Y}}}$$



- Batch용 Affine 계층 구현(Class 로 구현)



1 $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^{\mathrm{T}}$
(N, 2) (N, 3) (3, 2)

2 $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{Y}}$
(2, 3) (2, N) (N, 3)

3 $\frac{\partial L}{\partial \mathbf{B}} = \frac{\partial L}{\partial \mathbf{Y}}$ 의 첫 번째 축(0축, 열방향)의 합
(3) (N, 3)

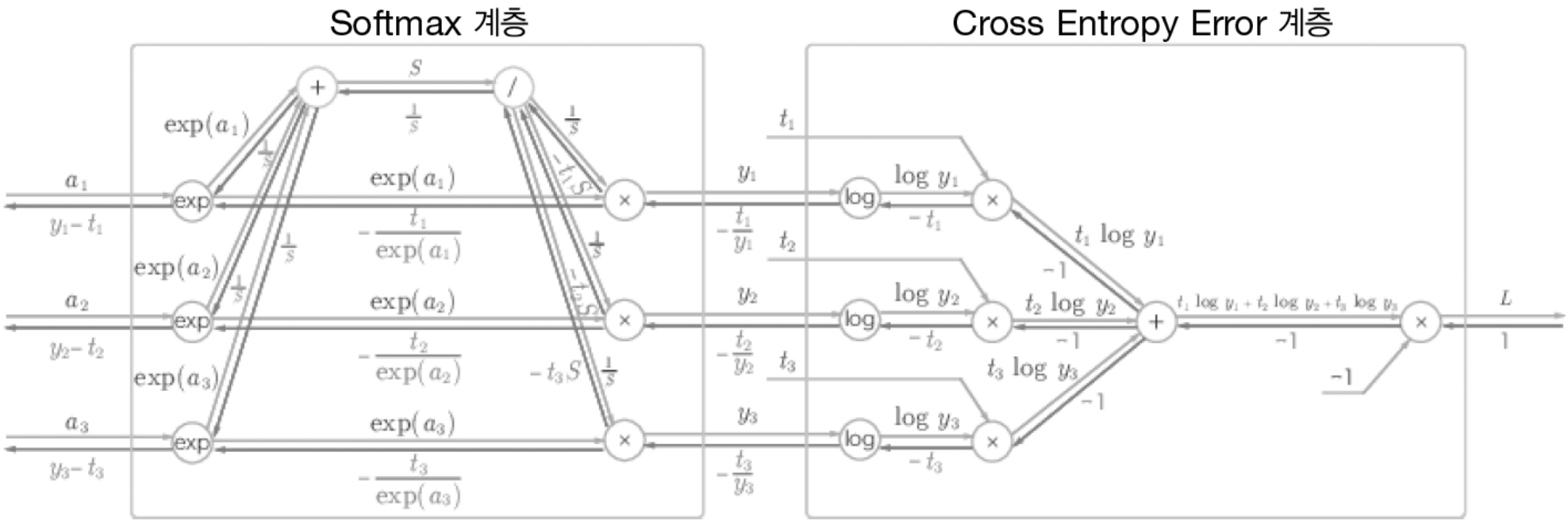
01. 활성화 함수 계층 구현

- Batch용 Affine 계층 구현(Class 로 구현)

```
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b
        self.x = None
        self.dW = None
        self.db = None

    def forward(self, x):
        self.x = x
        out = np.dot(x, self.W) + self.b
        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)
        return dx
```

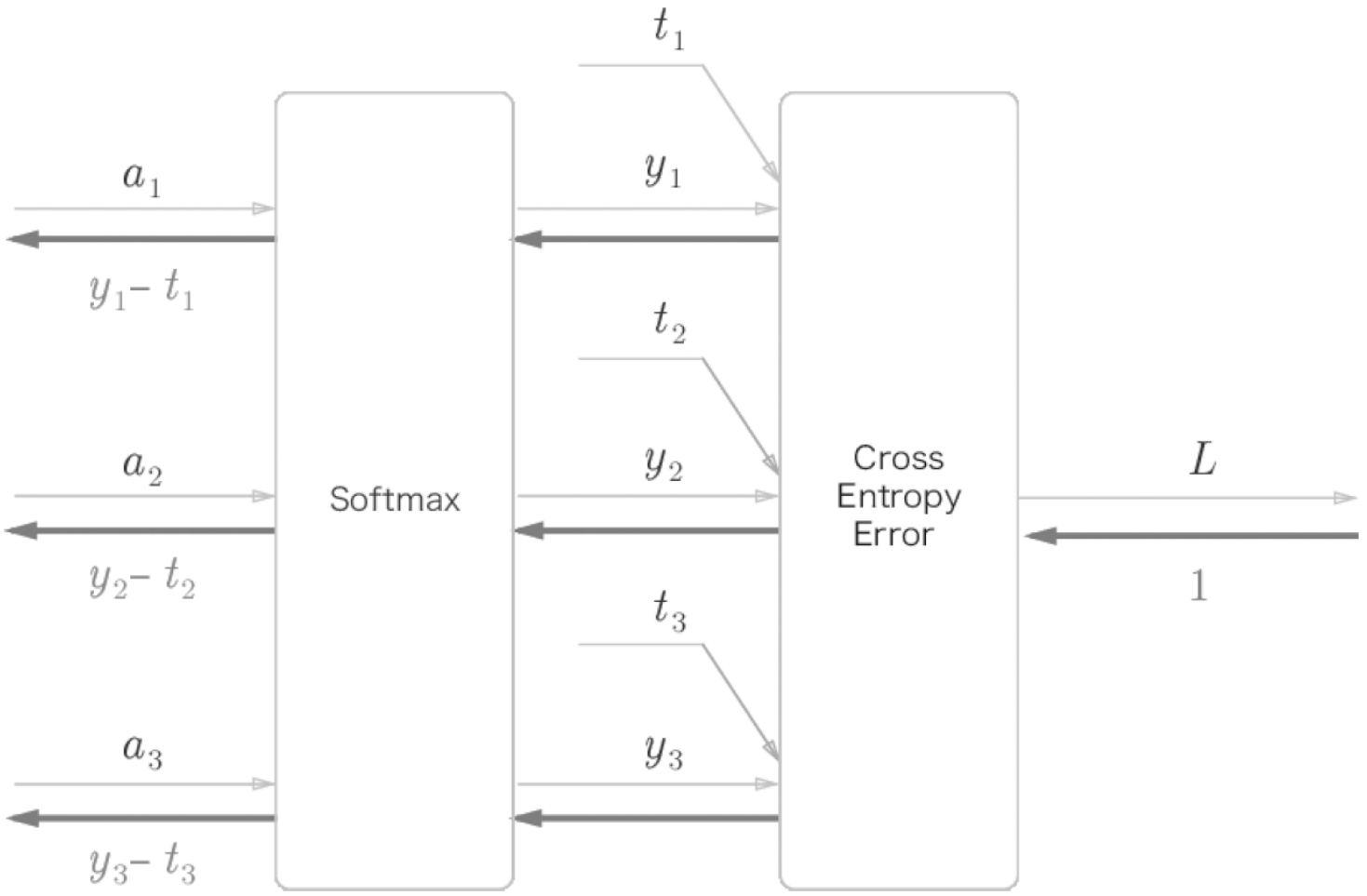
01. 활성화 함수 계층 구현

- Softmax-with-Loss 계층



01. 활성화 함수 계층 구현

- Softmax-with-Loss 계층



- Softmax-with-Loss 계층

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None
        self.t = None

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)
        return self.loss

    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        dx = (self.y - self.t) / batch_size
        return dx
```

02

오차역전파 검증

02. 오차역전파 검증

- 수치 미분을 이용한 방법과 오차역전파로 계산한 방법과의 비교
- 수치 미분의 경우 느리지만, 구현이 쉬워서 실수하기가 어려움
- 반면 오차역전파법은 빠르지만, 구현이 어려워서 실수가 많이 생길 위험이 있음
- 두 가지 방법의 비교를 통해 구현이 제대로 이루어졌는지 확인이 가능

02. 오차역전파 검증

- 수치 미분을 이용한 방법과 오차역전파로 계산한 방법과의 비교
- 둘 사이의 차이가 0이 되는 일은 잘 없음(계산의 정밀도의 한계가 존재)
- 거의 0에 가까운 값이 나오면 정상적인 구현

```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

x_batch = x_train[:3]
t_batch = t_train[:3]

grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)

# 각 가중치의 절대 오차의 평균을 구한다.
for key in grad_numerical.keys():
    diff = np.average( np.abs(grad_backprop[key] - grad_numerical[key]) )
    print(key + ":" + str(diff))
```

WeGo

02. 오차역전파 검증

- 기존 학습 코드의 기울기 계산 부분을 오차역전파법으로 변경

02. 오차역전파 검증

```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
```

02. 오차역전파 검증

```
for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    #grad = network.numerical_gradient(x_batch, t_batch) # 수치 미분 방식
    grad = network.gradient(x_batch, t_batch) # 오차역전파법 방식(훨씬 빠르다)

    # 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print(train_acc, test_acc)
```

WeGo

03

학습 관련 기술

WeGo

- 신경망 학습의 목적 → 손실 함수 값을 최소화하는 매개변수를 탐색 → 최적화
- 전체의 데이터가 아닌 확률적으로 추출하여 학습하는 SGD(Stochastic Gradient Descent)

$$W \leftarrow W - \frac{\eta \partial L}{\partial W}$$

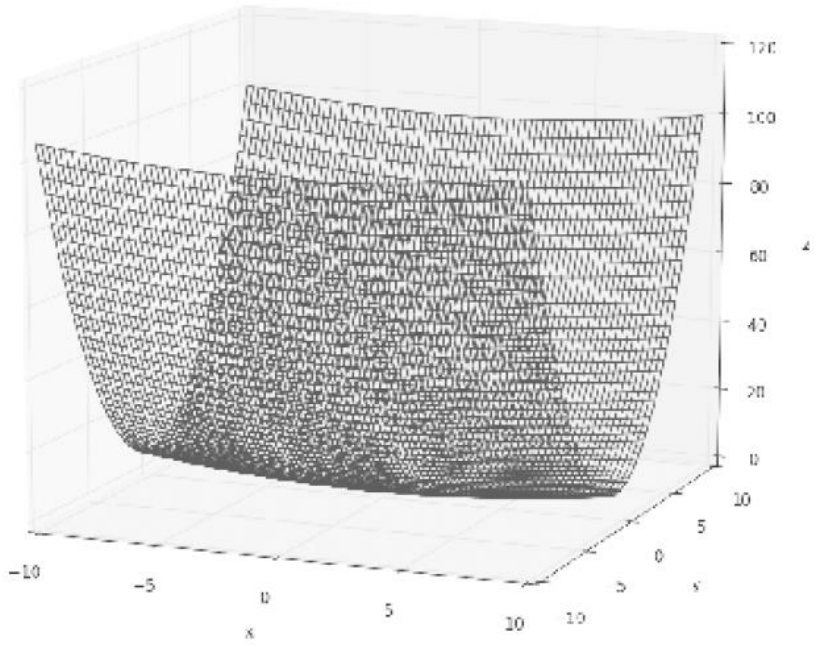
```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

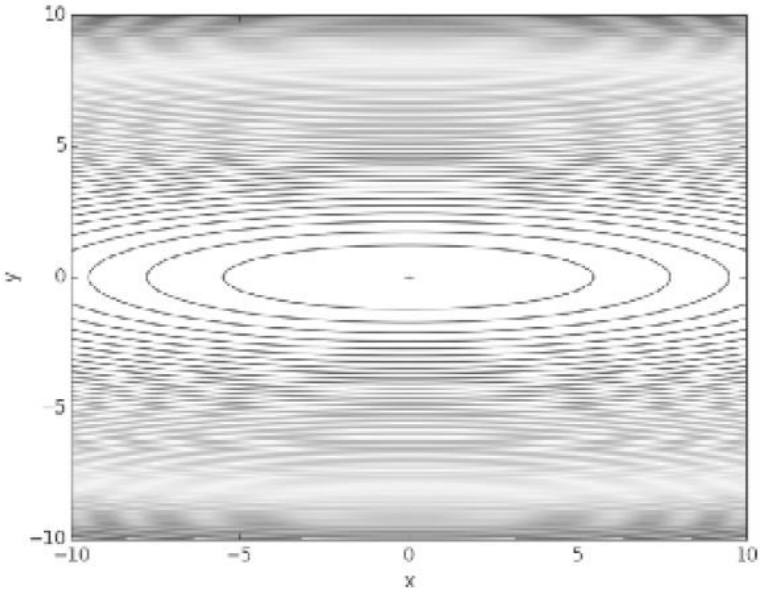
- 신경망 학습의 목적 → 손실 함수 값을 최소화하는 매개변수를 탐색 → 최적화
- 전체의 데이터가 아닌 확률적으로 추출하여 학습하는 SGD(Stochastic Gradient Descent)

```
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
optimizer = SGD()

for I in range(10000):
    …
    x_batch, t_batch = get_mini_batch(…)
    grads = network.gradient(x_batch, t_batch)
    params = network.params
    optimizer.update(params, grads)
    …
```

- SGD의 문제점

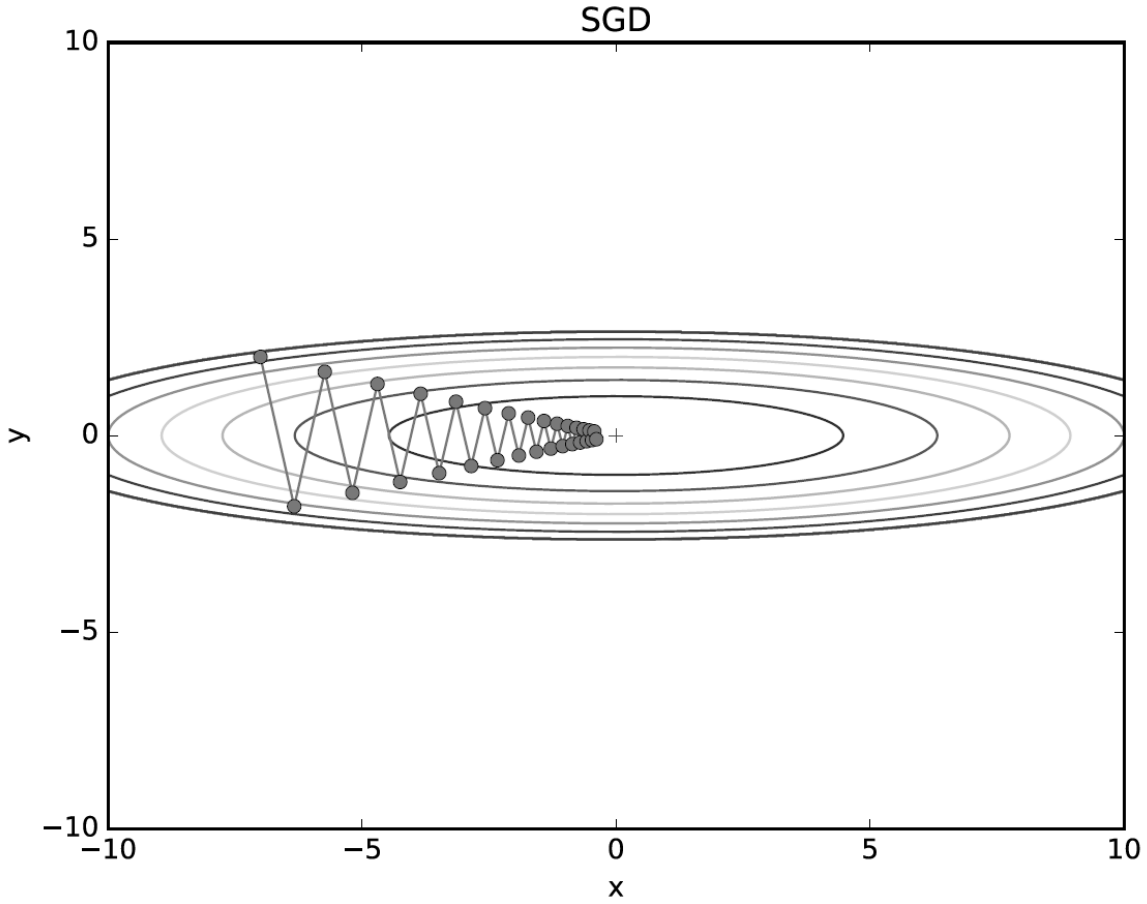
$$f(x,y) = \frac{1}{20}x^2 + y^2$$





- SGD의 문제점

$f(x, y) = \frac{1}{20}x^2 + y^2$에 대해 SGD 적용



- Momentum
- 기존의 개념에 대해 속도 가속도와 같은 개념을 추가
- V는 초기값이 0인 속도
- α는 가속도와 같은 역할(보통 0.9 등 1 이하의 값을 선택)

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \frac{\eta\partial L}{\partial W}$$
$$W \leftarrow W + \boldsymbol{v}$$

$$\frac{\partial L}{\partial W_1} = 5, \frac{\partial L}{\partial W_2} = 3$$
$$W \leftarrow W - \eta\frac{\partial L}{\partial W_1} = W - 0.5$$
$$W \leftarrow W - \eta\frac{\partial L}{\partial W_2} = W - 0.3$$

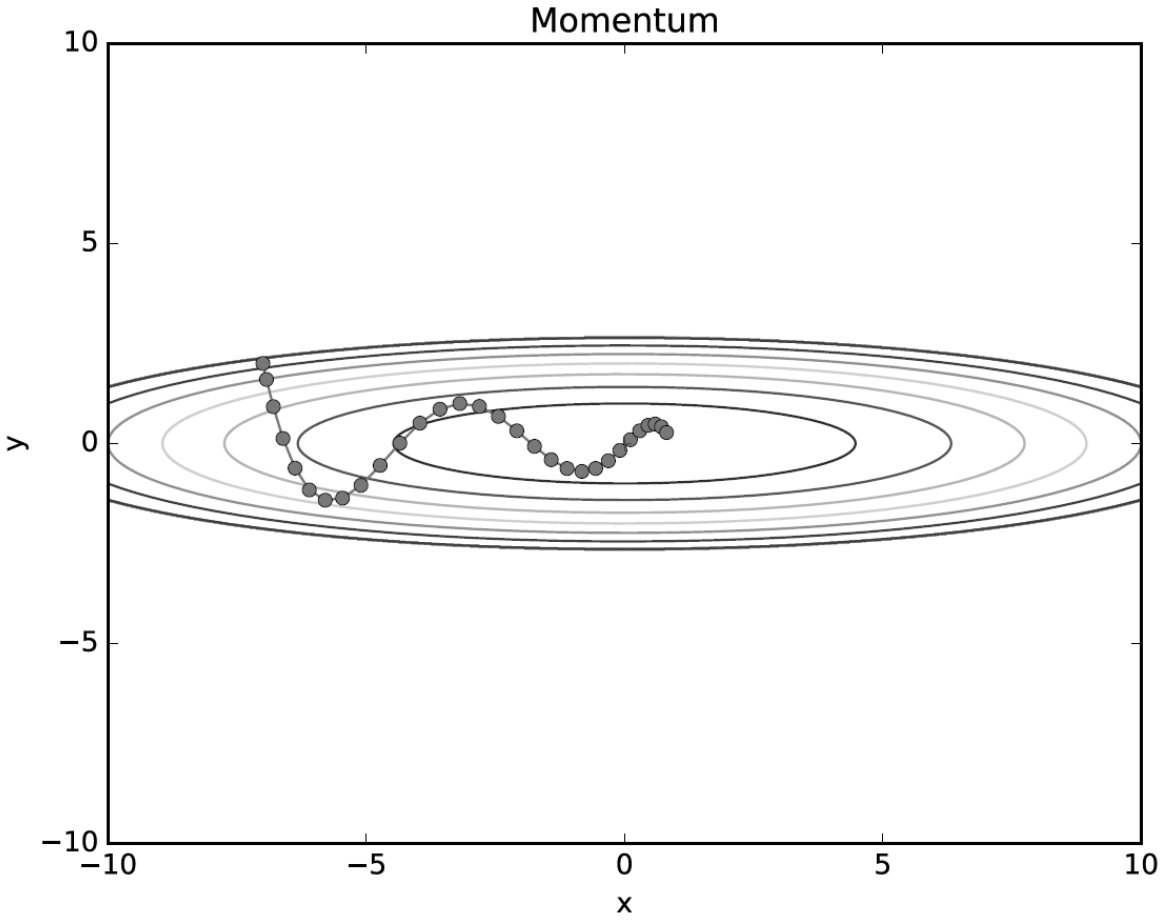
기존 방법

$$v_1 \leftarrow -\frac{\eta\partial L}{\partial W_1} = -0.5$$
$$W \leftarrow W + v_1 = W - 0.5$$
$$v_2 \leftarrow \alpha v_1 - \eta\frac{\partial L}{\partial W_2} = -0.45 - 0.3 = -0.75$$
$$W \leftarrow W + v_2 = W - 0.75$$

Momentum

- Momentum 적용

$$f(x, y) = \frac{1}{20}x^2 + y^2$$



- Momentum 구현

```
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momemtum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)
        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

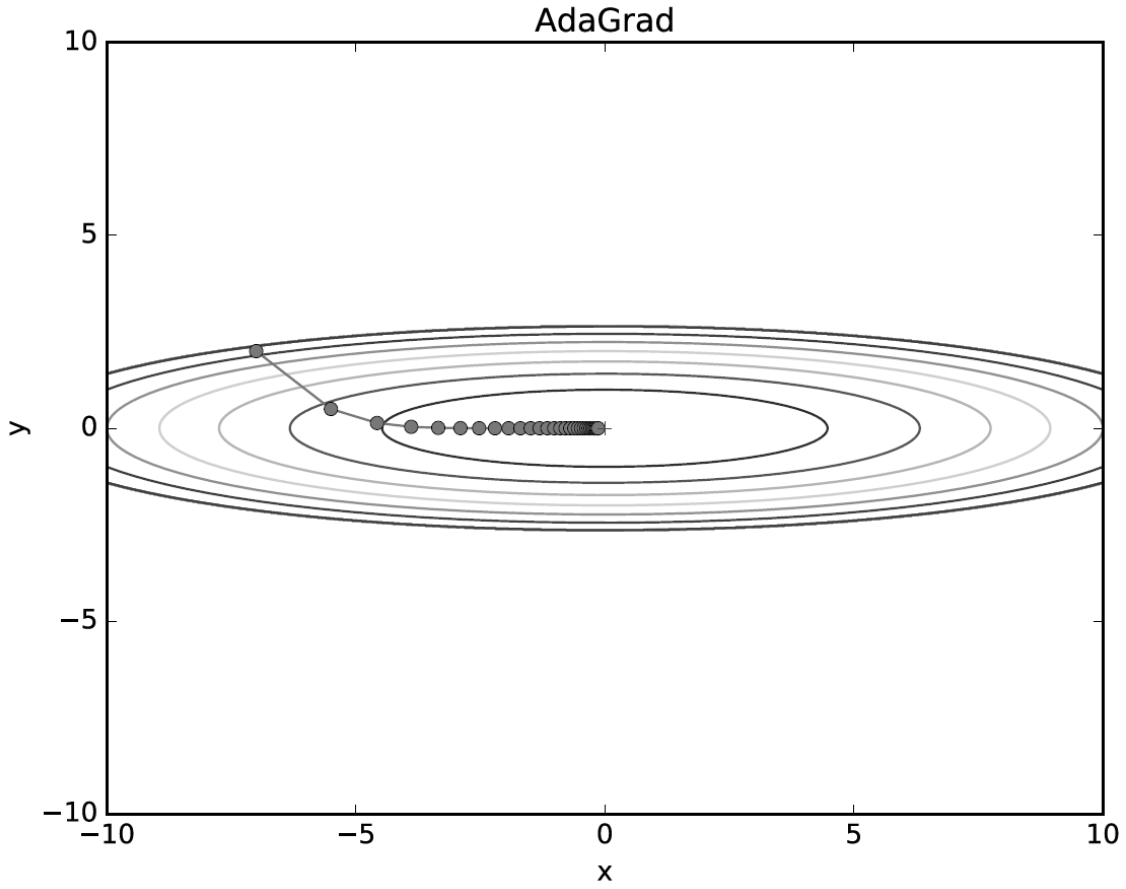
- AdaGrad
- 개별 매개변수에 적응적(Adaptive)으로 학습률을 조정하면서 학습을 진행

$$\boldsymbol{h} \leftarrow \boldsymbol{h} + \frac{\partial L}{\partial \boldsymbol{W}} * \frac{\partial L}{\partial \boldsymbol{W}}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{1}{\sqrt{\boldsymbol{h}}} \frac{\partial L}{\partial \boldsymbol{W}}$$

- AdaGrad 적용

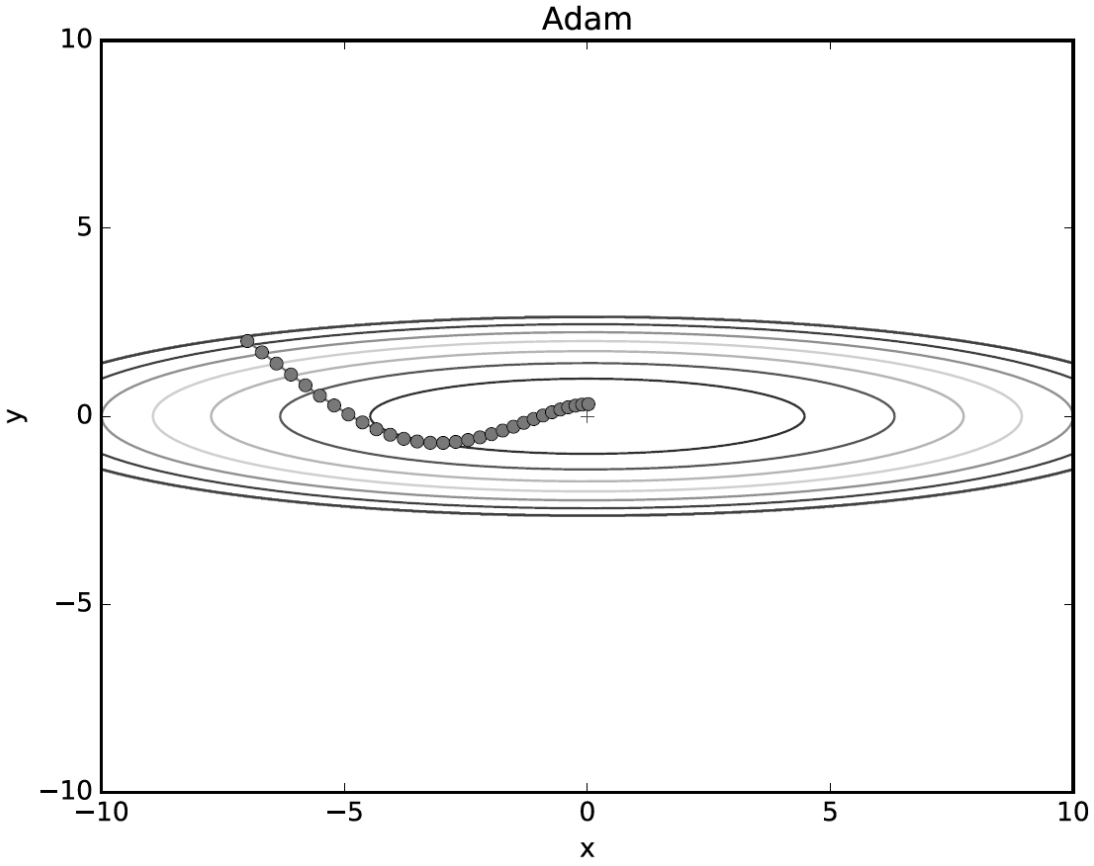
$$f(x,y) = \frac{1}{20}x^2 + y^2$$



- AdaGrad 구현

```
class AdaGrad:
    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)
        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

- Adam
- 위의 두 기법을 융합한 방법



- AdaGrad 구현

```
class Adam:
    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None
```

- AdaGrad 구현

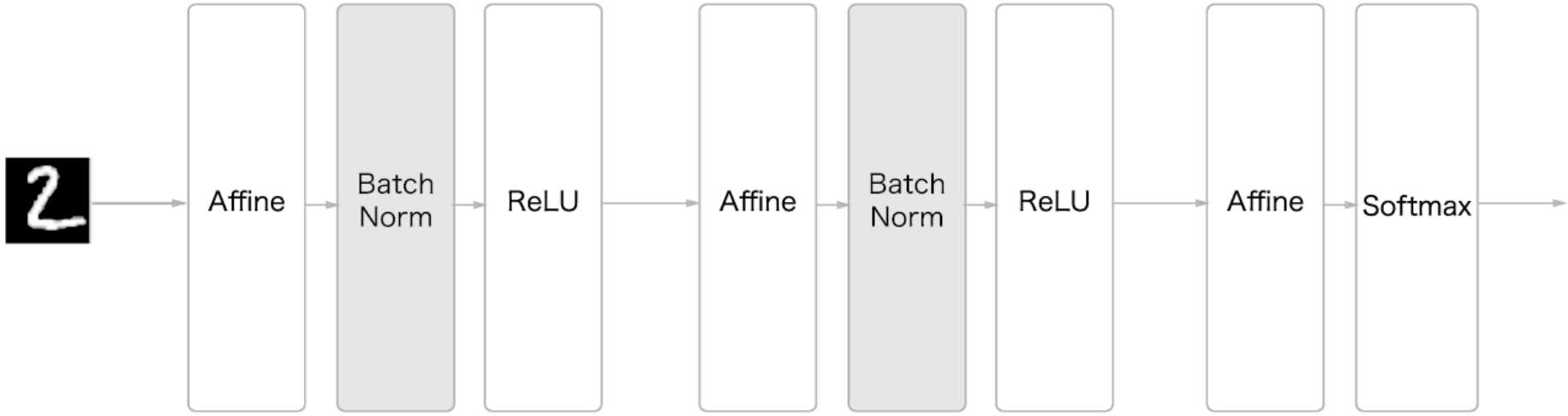
```
def update(self, params, grads):
        if self.m is None:
            self.m, self.v = {}, {}
            for key, val in params.items():
                self.m[key] = np.zeros_like(val)
                self.v[key] = np.zeros_like(val)

        self.iter += 1
        lr_t  = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

        for key in params.keys():
                        self.m[key] += (1 - self.beta1) * (grads[key] - self.m[key])
            self.v[key] += (1 - self.beta2) * (grads[key]**2 - self.v[key])

            params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)
```

- 배치 정규화
- 학습 속도를 개선할 수 있고, 초기값에 크게 의존하지 않으며, 오버피팅을 억제
- 미니 매치를 평균이 0, 분산이 1이 되도록 정규화를 진행

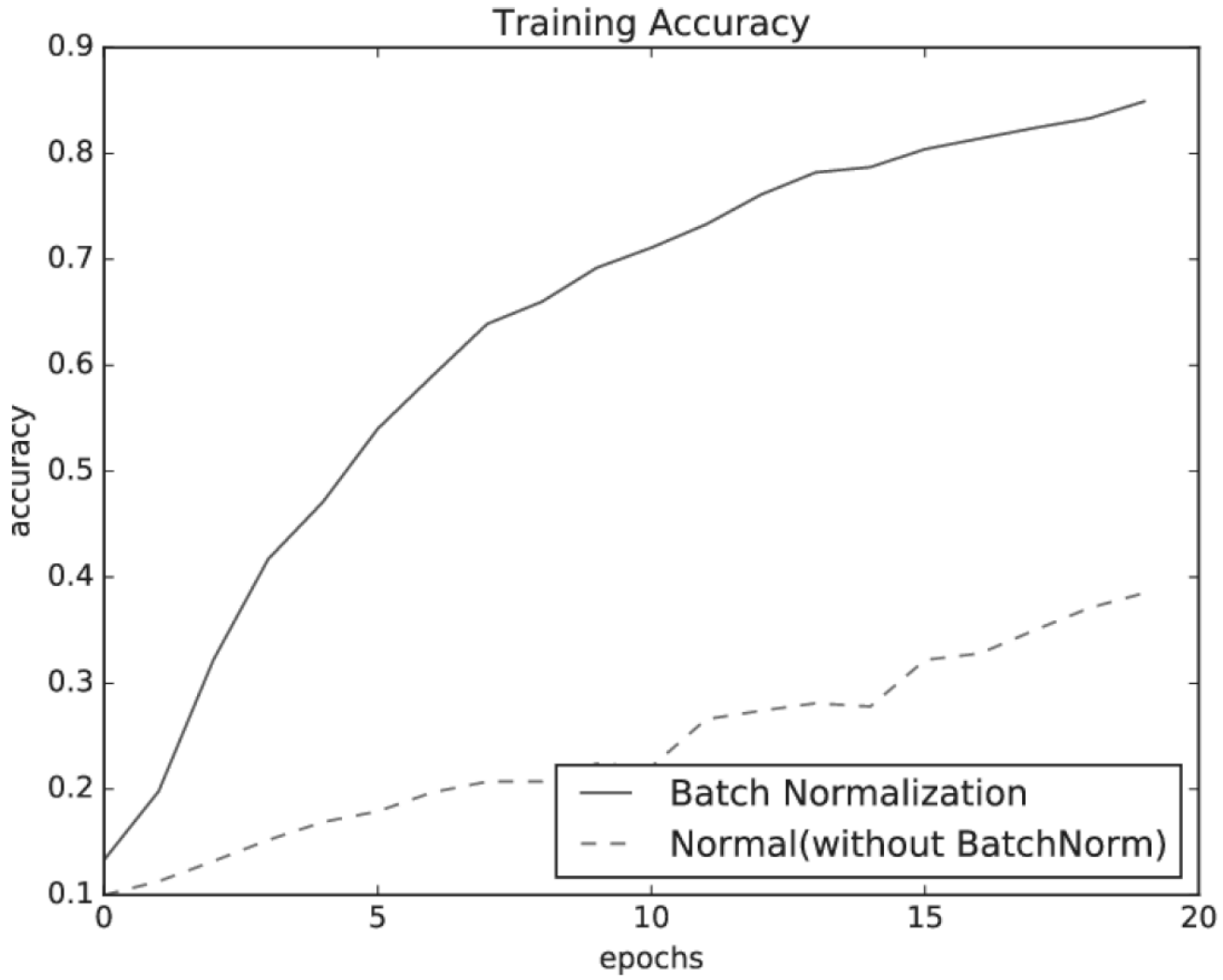
$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \quad \sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\widehat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

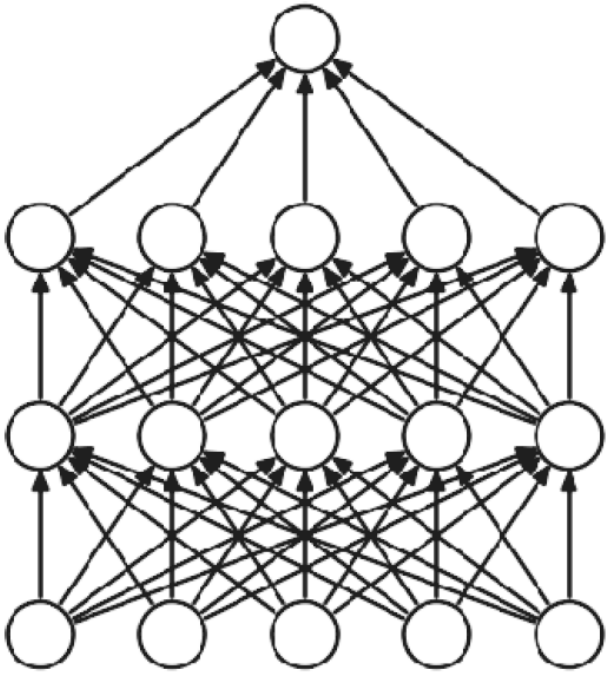
- 배치 정규화
- 학습 속도를 개선할 수 있고, 초기값에 크게 의존하지 않으며, 오버피팅을 억제
- 미니 매치를 평균이 0, 분산이 1이 되도록 정규화를 진행



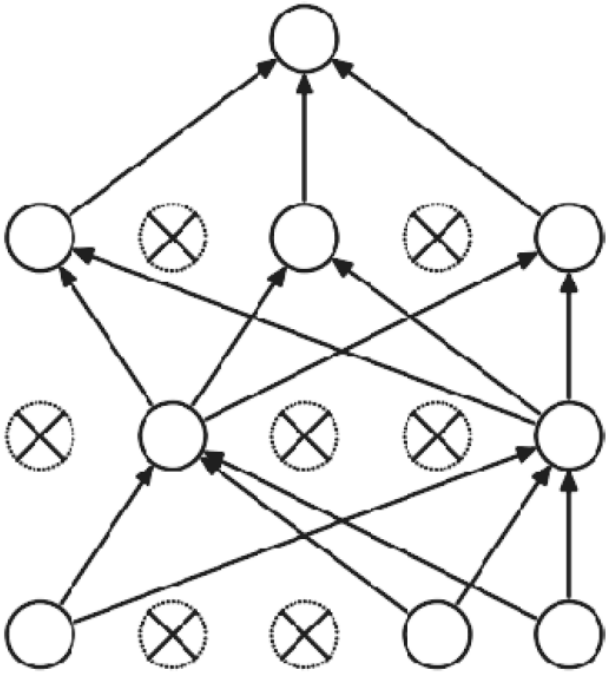
- 배치 정규화에 따른 학습 속도



- 드롭 아웃
- 뉴런을 임의로 삭제하면서, 오버피팅을 억제하는 방식

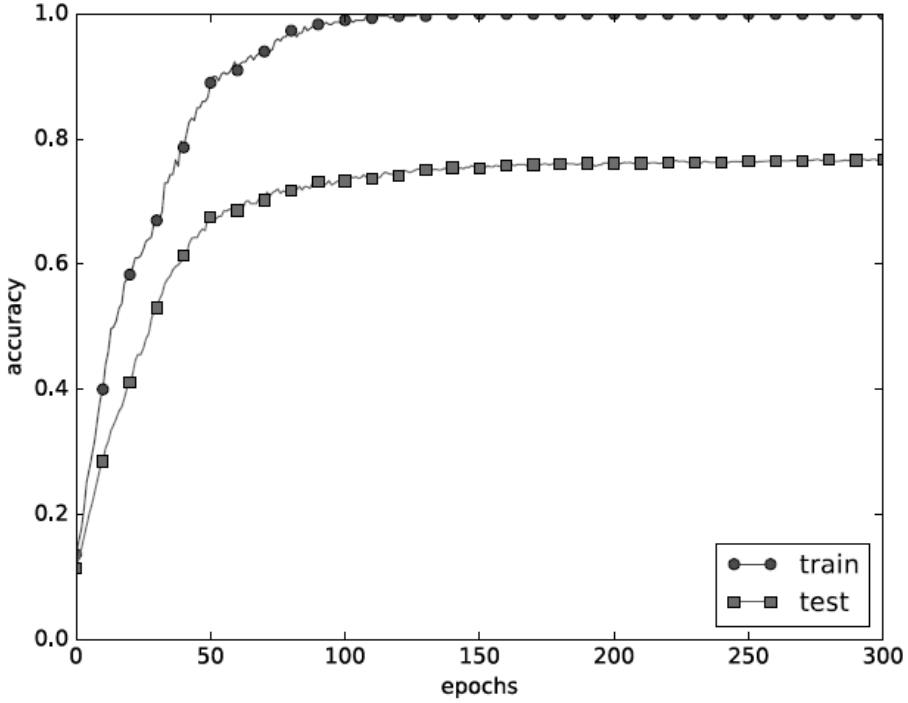


(a) 일반 신경망

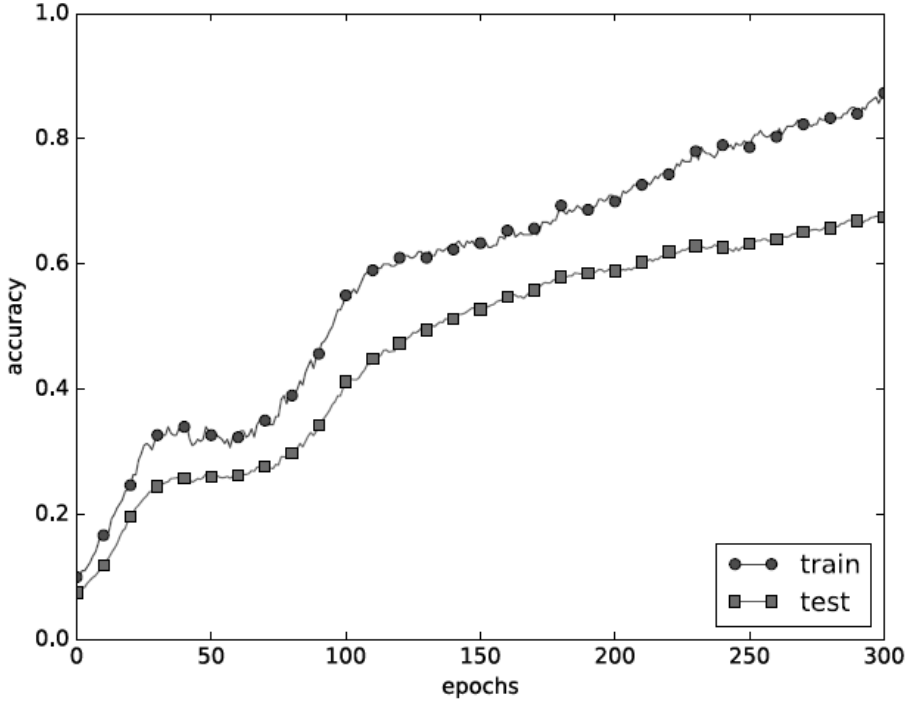


(b) 드롭아웃을 적용한 신경망

```
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 – self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
```


1.0
0.8
0.6
0.4
0.2
0.0
accuracy
0
50
100
150
200
250
300
epochs
train
test

일반


1.0
0.8
0.6
0.4
0.2
0.0
accuracy
0
50
100
150
200
250
300
epochs
train
test

드롭 아웃 적용

WeGo